CS354: Machine Organization and Programming

Lecture 26
Monday the November 02nd 2015

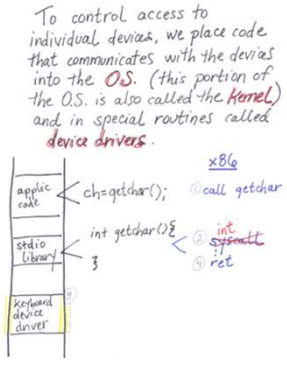
Section 2
Instructor: Leo Arulraj

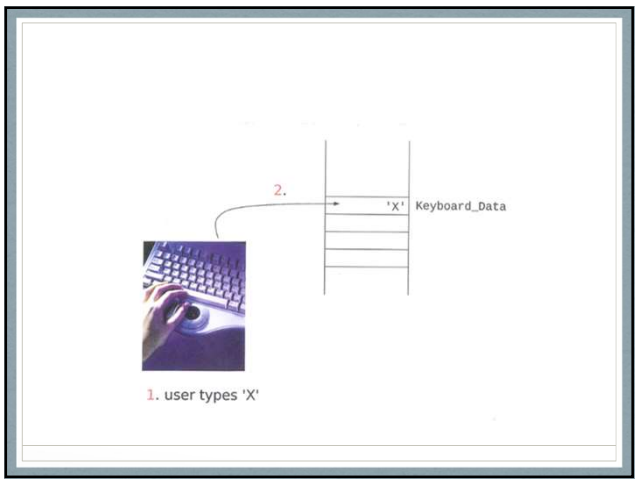
Class Announcements

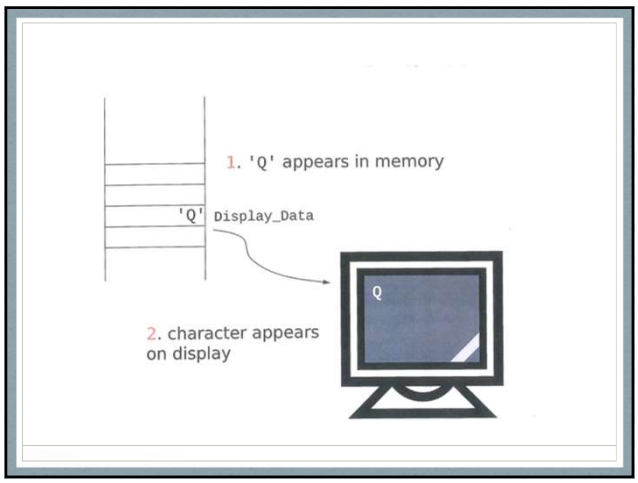
1. Programming Assignment 3 is due by 9 AM day after tomorrow (11/04 – Wednesday). As usual, you can submit it upto 48 hours after the deadline with penalties.

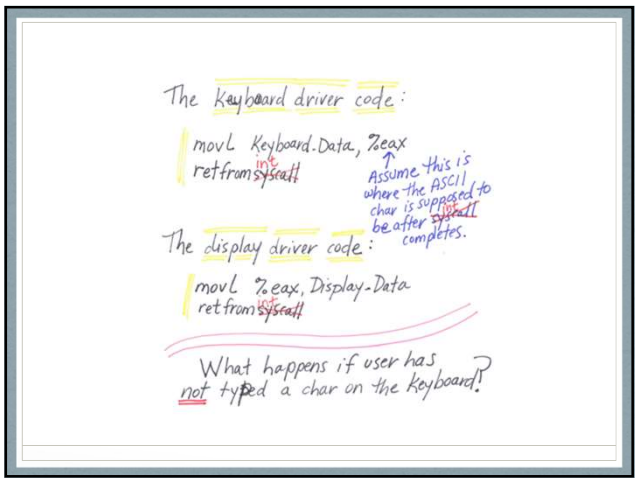
Lecture Overview

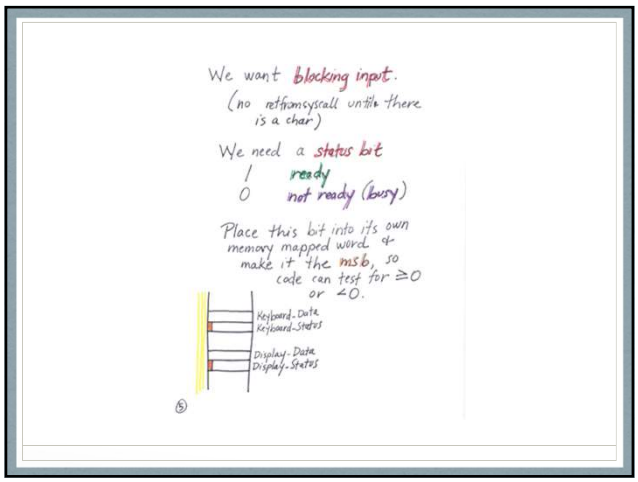
1. Interrupts and Exception
2. Intro to Processes

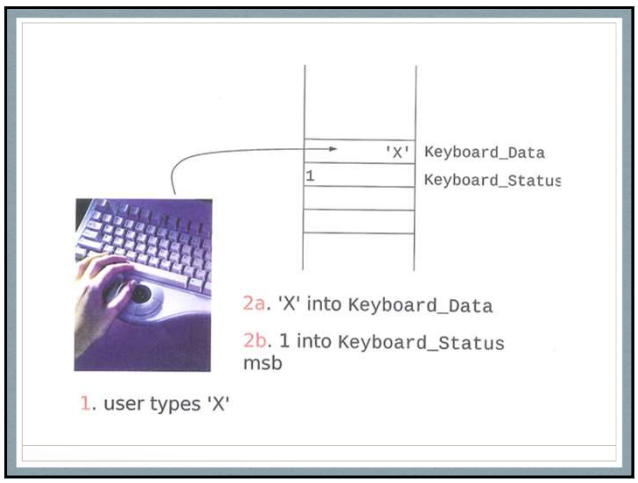


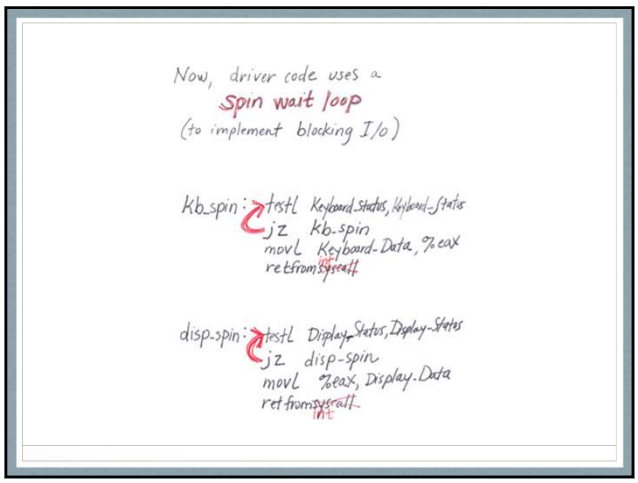

2.
'X'
Keyboard_Data
1. user types 'X'

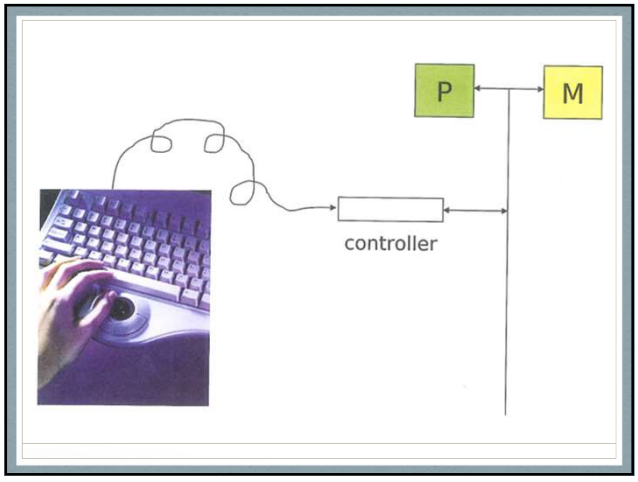

1. 'Q' appears in memory
'Q'
Display_Data
Q
2. character appears on display

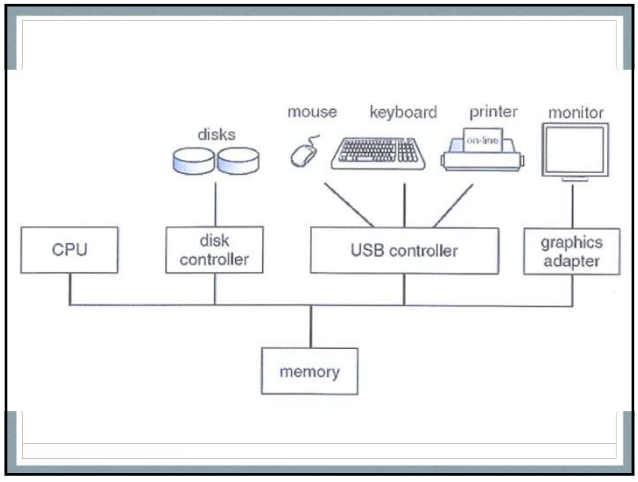

The Keyboard driver code:
movl Keyboard_Data, %eax
int
retfromsyscall
Assume this is where the ASCII char is supposed to be after syscall completes.
The display driver code:
movl %eax, Display_Data
int
retfromsyscall
What happens if user has not typed a char on the keyboard?

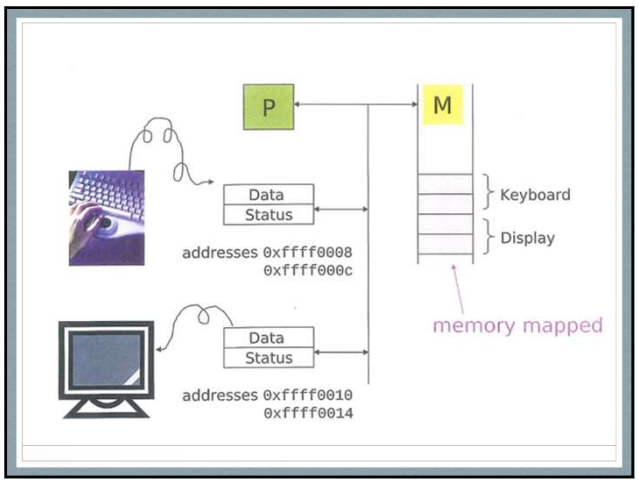

We want blocking input.
(no retfromsyscall until there is a char)
We need a status bit
1 ready
0 not ready (busy)
Place this bit into it's own memory mapped word. or make it the msb, so code can test for ≥0 or <0.
Keyboard_Data
Keyboard_Status
Display_Data
Display_Status
15


'X'
Keyboard_Data
1
Keyboard_Status
2a. 'X' into Keyboard_Data
2b. 1 into Keyboard_Status msb
1. user types 'X'


Now, driver code uses a
spin wait loop
(to implement blocking I/O)
kb_spin: testl Keyboard_Status, Keyboard_Status
jz kb_spin
movl Keyboard_Data, %eax
ret from syscall
disp_spin: testl Display_Status, Display_Status
jz disp_spin
movl %eax, Display_Data
ret from syscall


P
M
controller


disks
mouse
keyboard
printer
monitor
on-line
CPU
disk controller
USB controller
graphics adapter
memory

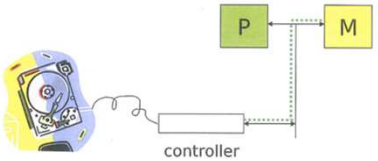


Byte transfers are OK,

But, what about faster devices that like to transfer more than a byte ?

the solution: DMA

Direct Memory Access



Issue for spin wait loop implementations:

One byte *only* in `_Data` has the potential for an incorrect result.

For example, if the user types 2 characters on the keyboard before `getchar()` is called.

The needed fix introduces a kernel-maintained queue for each device.

Then, the kernel polls to check status bits and handle any ready devices.

INTERRUPTS

here is an analogy. . .

Teacher is OS

each student is I/O device

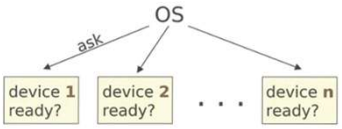
Consider the *inefficiency* of OS polling.

1

INTERRUPTS

Because polling is *so inefficient*,

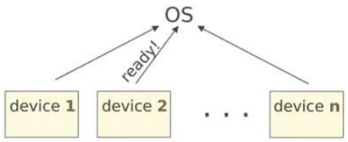
instead of



2

INTERRUPTS

Turn the situation upside down



3

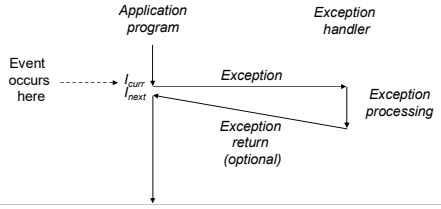
INTERRUPTS

4

Anatomy of an exception

An exceptions is an abrupt change in control flow.

Examples: div by 0, arithmetic overflow, page fault, I/O request completes, Ctrl-C



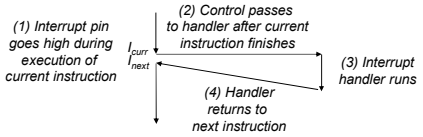
Classes of Exceptions

1. Interrupts (Asynchronous): Always return to next instruction.
2. Traps & System Calls (Synchronous): Always return to next instruction.
3. Faults (Synchronous): Might return to next instruction.
4. Aborts (Synchronous): Never returns

Interrupts

Examples of Interrupts:

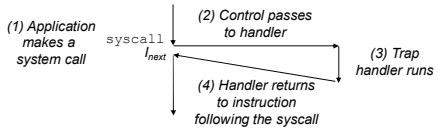
- Timer interrupt
- Arrival of a packet from a network
- When a key is pressed on the keyboard
- When the mouse is moved



Traps

Traps are intentionally issued by executing an instruction.

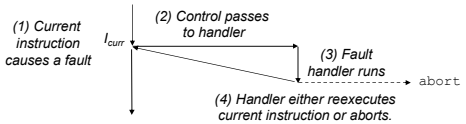
Example: System calls



Faults

Faults result from error conditions that might be correctable.

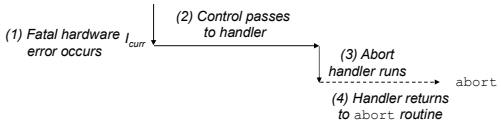
Examples: Page fault, Divide error



Aborts

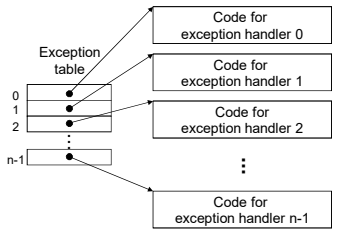
Aborts result from unrecoverable fatal errors.

Example: parity errors due to DRAM bit corruption



Exception Table

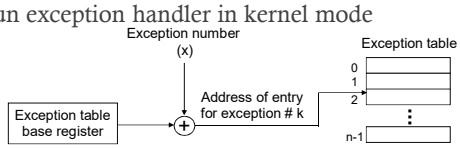
Exception table



Exception Table lookup

Exception is similar to procedure calls except for some important differences:

- Return address is not the next instruction always
- Push EFLAGS register also onto kernel stack
- Run exception handler in kernel mode



IA32 Exception Table

From CSAPP text book:

Exception Number	*Description*	*Exception Class*
0	Divide error	Fault
13	General protection fault	Fault
14	Page fault	Fault
18	Machine check	Abort
32-127	OS-defined	Interrupt or trap
128 (0x80)	System call	Trap
129-255	OS-defined	Interrupt or trap

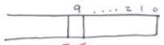
Rather important, but not covered in textbook:

If running a handler, and a new interrupt request arrives, what should happen?

* Continue on, complete handling of current interrupt, then, when done, deal with new request?
  (probably) nonreentrant
* Interrupt the handling of this interrupt?
  reentrant

Every architecture has a control bit which identifies whether the F+E cycle is paying attention to IRQs. Called interrupt enable

on x86:
EFLAGS



fetch+execute cycle:
1 enabled
0 disabled

0. if IF=1 and interrupt requested, go handle it
1. fetch instr
2. PC update
3. decode

Consider the x86 instruction:

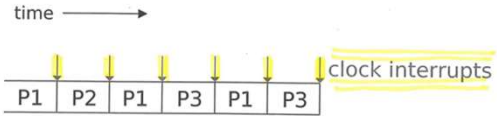
`cli` clear IF

What happens if an application includes this `cli` *instruction?*

Irrelevant (to this discussion) x86 instruction:

`sti` set IF

OS relies on
clock interrupts
to allocate processing time. . .



As the clock interrupts, the kernel runs, and it decides which program runs next.

Clarified instruction:

`cli` clear IF *if CPL is high enough, otherwise trap*

Does `sti` also need to be a privileged instruction?

Keep IF=1 while CPL=00.
(So, applications can always be interrupted.

HW must disable interrupts while saving state & at least until first instruction within handler is fetched.

Better Definitions:

nonreentrant IF=0 the entire time a handler runs

reentrant interrupts may be reenabled while handler runs (usually only for higher priority requests)